to perform other tasks, its context is saved so that it can be restored later before resuming execution of the code. This mechanism allows the code to resume running and proceed as [thought] though it had not been interrupted, which can help create the illusion of more than one piece of code executing simultaneously on a single CPU 110. On a host computer 100 having several CPUs 110-110b, a piece of code might be executed on one CPU for a while, interrupted, and then resumed on a different CPU without breaking the illusion of continuity. A single host computer may have several process contexts, but there can be only one process context that is active per CPU at a given point in time. This active process context is the software environment for the code currently running on the CPU. A process context is never active on more than one CPU concurrently.

Please replace the paragraph beginning on page 12, line 18, with the following rewritten paragraph:

A passive software component is designed to run periodically in a process context created for some other, active component. Passive software component instances are typically loaded into address spaces created for active [opponents] components. After initialization, they run only when another component transfers control to them using a call instruction. Typically, passive component instances perform defined tasks each time they are called, and then they return control to the calling component. A passive component can call other passive components. The target of a call is a function (i.e. to transfer control to a piece of code using a call instruction is known as calling a function.) Each function is a piece of code that performs a defined task, and then returns control to the code that executed the call instruction. When control is returned, the calling code continues execution beginning with the first instruction after the call instruction.

Please replace the paragraph beginning on page 17, line 11, with the following rewritten paragraph:

Turning to Figure 3, Figure 3 is a flowchart illustrating a process 300 for the scheduling driver to start an I/O request and to provide an estimated processing time for the completion of an I/O transaction to an application, according to one embodiment of the invention. Upon the entry point (block 310), the process 300 determines if the locked flag is set (block 315). If the locked flag is set, the process 300 calculates an estimated amount of time left (EATL) until the device will be free again (block 320). Thus, the device is currently busy servicing another request. The estimated amount of time left (EATL) is calculated by subtracting the elapsed time since the current request being processed was started from the requests original estimated processing time (EPT). If the EATL calculation is negative, the EATL calculation is set to zero (block 320). Next, the process 300 provides the estimated amount of time left (EATL) until the device will be available to the application (block 325). The process 300 then returns a "busy" signal to the process 200 and goes to block 215 of the process 200 illustrated in Figure 2 (block 330).

Please replace the paragraph beginning on page 20, line 19, with the following rewritten paragraph:

The processes for the interaction of applications with the scheduling driver provide an elegant way for many competing applications to fairly share access to a device that does not generate interrupts. Further, the invention achieves this without causing system performance problems by using an estimate of how long the device will take to complete each request. Since all waiting [application] applications become runnable on the same timer tick, the scheduling driver schedules them in the same order that they would be scheduled if they were all blocked waiting for the device to become available. However, this assumes that the OS scheduler fairly picks the next process context to run from among a group of process contexts that have just become runnable at the same time. Thus, the fairness of scheduling driver is also based on the

fairness of the process context scheduling by the scheduler of the operating system. For example, the invention works well with multilevel round robin schedulers used by Microsoft Windows operating systems (e.g. Windows 98, windows NT 4.0, Windows 2000) that operate fairly. Generally, the invention is best suited to work with operating systems in which applications generally remain at the same priority level and where a particular priority level runs in round robin order. However, it should be appreciated that the present invention can also work well with a wide variety of operating system schedulers that utilize differing priority schemes.

Please replace the paragraph beginning on page 22, line 22, with the following rewritten paragraph:

The processes for the interaction of applications with the scheduling driver, according to one embodiment of the present invention, provide an elegant way for many competing applications to fairly share access to a device that does not generate interrupts. The invention achieves this without causing system performance problems by using an estimate of how long the device will take to complete each request. Further, the invention can work with most operating systems. It does no blocking at all in the scheduling driver- all blocking is done in the application code. Additionally, even though device request synchronization is done in different ways on different operating systems, the processes utilized in the present invention tend to make it very portable.

IN THE CLAIMS

Please cancel claims 4, 18, and 32.

Following is a complete set of claims as amended with this response, which includes amendments to claims 1, 5, 15, 19, 29, and 33.